IEEE COPYRIGHT
In Proceedings of the 5th IEEE International Conference on Electronics, Circuits and Systems, Lisbon, Portugal, September 7-10, 1998.

Towards an Automatic Path from Java™ *Bytecodes* to Hardware Through High-Level Synthesis[1]

João M P Cardoso
Universidade do Algarve/INESC
Phone: +355 1 3100288

Horácio C Neto
IST/INESC
Phone: +355 1 3100366

INESC, Rua Alves Redol 9, 1000 Lisboa, Portugal
{Joao.Cardoso, hcn}@inesc.pt
FAX: +351 1 3145843

Abstract

This article describes a new approach to synthesise dedicated hardware from a system specification using the Java language. The new compiler named GALADRIEL starts from Java *classfiles* produced from the initial Java specification and processes the system information in order to exploit the concurrency implicit in each method, so that it can be efficiently implemented by multiple hardware and/or software components. The paper gives emphasis to the compiler techniques used to exploit the implicit concurrency and to the use of high-level synthesis to generate the hardware models from the Java *bytecodes* information.

1. Introduction

The Java language [1] has features that make it suitable to be adopted as an embedded systems specification language [2], as a generic programming language [3], and also as the Internet language. It has syntax mechanisms to support concurrency (critical to embedded systems), is architectural independent ("write once, run everywhere"), is a simple object-oriented language, is strongly typed, contains exception handling mechanisms, is safe and robust, is dynamic, etc.

However it has a few drawbacks, namely the low performance of the execution code. Some efforts to approximate the execution time of Java programs to similar C/C++ programs take advantage of: optimisation techniques integrated in JIT (**J**ust-**I**n-**T**ime) compilers [4], optimisation of *bytecodes* with advanced compiler techniques [5][6], loop parallelization [7] when real multithreaded is supported, and development of direct execution support (Java microprocessors) [8]. Nevertheless all of these efforts can be substantially improved with the integration of HW/SW codesign and **H**igh-**L**evel **S**ynthesis (**HLS**) [9] techniques. Typical HW/SW codesign approaches start from a SW specification and migrate SW blocks to HW in order to satisfy timing constrains [10] or to accelerate a generic application in a host by means of HW boards [11]. Others begin with an HW specification and migrate to SW some of the HW blocks such that the timing requirements of the specification remain satisfied [12].

Our approach starts with an abstract model of computation and uses co-synthesis techniques supported by an **HLS** system to accelerate the Java applications. The front-end is the GALADRIEL compiler, which explores the parallelization of SW blocks, and the translation from *bytecodes* to VHDL.

This paper describes some of the work done so far and indicates future directions for the research involved. In Section 2 we summarise the Java technology. Then, Section 3 describes the computations and transformations performed by the compiler. The automatic path from the specification to the HW implementation is described in Section 4. Finally in Section 5, an example is presented and the results obtained are shown.

2. The Java Technology

Each Java class of a given application is compiled to a virtual machine designated by **JVM** (**J**ava **V**irtual **M**achine) [13]. The compilation results of each Java class reside in a file named Java *classfile* which contains a constant pool (with symbolic references, constant strings, and information about the class) and the *bytecodes* for each method, among other information [13]. The **JVM** has about 200 instructions that can be distributed over 30 functionality type groups and each instruction has an 8-bits opcode. The **JVM** is stack-oriented, with an operand's stack and local variables for each method. The operations get values from the operand stack and store the result in it. The operand stack is also used to pass arguments to methods and to receive the return result from a called method. There are instructions to transfer word contents from/to the local variables to/from the operand stack, instructions related to the manipulation and creation of objects, arrays, and the information encapsulated within it, control-flow instruc-

[1] This work has been partially supported by the Ph.D. program of the Prodep 5.2 action and the program Praxis XXI under the scope of Project PRAXIS/2/2.1/TIT/1643/95.

IEEE COPYRIGHT
In Proceedings of the 5th IEEE International Conference on Electronics, Circuits and Systems, Lisbon, Portugal, September 7-10, 1998.

tions (like *if*, *goto*, and *switch* types), jumps to subroutines handling exceptions, method invocation, etc. The **JVM** supports multithreaded applications by means of two *bytecode* instructions to synchronise blocks of code: *monitorenter* and *monitorexit*. Additional support is provided by the *java.lang.Thread* class [1]. Object access synchronisation across multiple threads is specified with the word *synchronised* and treated with the two previous *bytecode* instructions.
The *classfiles* can be executed using an interpreter (like *java* from Sun™) or a JIT compiler (compiles on-the-fly *bytecodes* to native code); directly with a Java microprocessor; or with an off-line compilation to native code.

3. The GALADRIEL Compiler

From the set of *classfiles* of a given application the GALADRIEL extracts the needed information and represents it in an internal model that contains a **DAG** (**D**irected **A**cyclic **G**raph), where each node represents a method and each edge represents the method invocation hierarchic flow (see Fig. 1). This graph is called **M**ethod **C**alling **S**equence **G**raph (**MCSG**) and its root is the *main()* method. Each *run()* method in the **MCSG**, which correspondent class extends the *Thread* class or implements the *Runnable* interface, is tagged as concurrent.

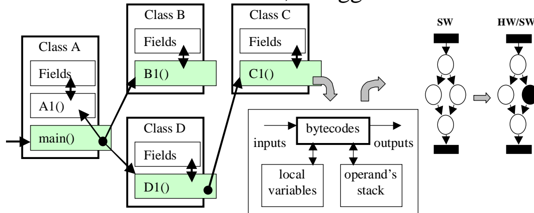

Fig. 1. The Method Calling Sequence Graph and the methodology flow.

From the *bytecodes* of each method in the **MCSG**, the GALADRIEL extracts the instructions and performs the transformations shown in Fig. 2. For each method a control flow graph (**CFG**) [14], G=(V, E), is generated that consists in a directed graph with each vertex V representing a **JVM** instruction and edges E representing the execution dependence between instructions. Each **CFG** includes a START and an END vertex (all other vertices are included in a path between the two vertices). There is an edge between two vertices (V_1, V_2) if and only if the instruction in V_2 can be executed immediately after the instruction in V_1. This model of **CFG** includes switch type vertices (like the *tableswitch* and *lookupswitch* of the **JVM** [13]). The compiler may optionally output to a file an ascii representation of each graph to be viewed by the DOTTY tool [15].
To analyse the data and control dependencies of a method at basic block level, the **CFG** after the computation of the leaders is transformed in the **CFG**$_{BB}$, in which the new vertices are basic blocks (**BBs**) [14] of **JVM** instructions.

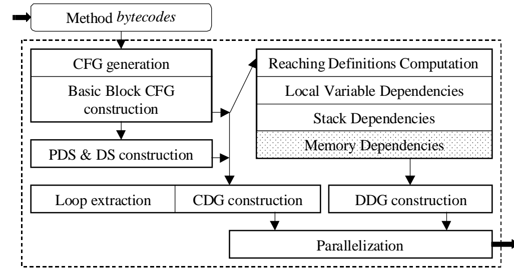

Fig. 2. Intra-procedural compiler phases.

With this graph the dominators set (**DS**) is computed by the iterative algorithm described in [14]. From that the compiler identifies the back edges, the **BBs** forming natural loops, and groups them in a special vertex after the identification of the related loops information (bounded, unbounded, counter variable, incremental value, initial value, and final condition). In the presence of nested loops the set of basic blocks of an outer loop contains the equivalent set of the inner loop and so on. Like in [16] the **CDG** (**C**ontrol **D**ependence **G**raph) is created from the **CFG**$_{BB}$ and the pos-dominators set (**PDS**). The **PDS** is constructed with the same algorithm of the **DS** in the reverse traversing order of the **CFG**$_{BB}$.
To construct the **DDG** (**D**ata **D**ependencies **G**raph) the GALADRIEL first determines the *reaching definitions* [14], from the local variables accessed by the **JVM** instructions in order to compute the data-dependencies over local variables, and then computes the stack-dependencies (**SD**). From that it computes the **U**se-**D**efinition (**UD**) chain for each use of a local variable that indicates whose local variable's definitions reach an usage. From the **UD** it constructs the **D**efinition-**U**se (**DU**) chain for each definition of a local variable, that indicates the set of usages that are reached by each definition of a variable.
A **BB**$_i$ is data-dependent of **BB**$_j$ if it uses a local variable defined by the last time in **BB**$_j$. The **SD** are determined traversing the instructions and storing for each one the state of the stack that consists in the information about the stack contents (operands, types, instruction that push the operand, etc.). This is possible because the stack model of the **JVM** is invariant [13], i. e. the stack contains always the same number and type of operands in the execution flow of each instruction of the **CFG**. After this, for each **BB** our algorithm only examines the stack-state of each instruction in the **BB** to determine its dependencies. If all the stack-states that enter in a **BB** exit it then this **BB** does not depends of another **BB** unless any instruction that uses stack operands without pop them (like the *dup* instruction) is involved. In this case the algorithm examines the operands involved. The data-dependencies caused by an utilisation of a variable or stack-operand that stores a constant do not generate any data-dependencies indeed. The **DDG** is the union of the two generated dependencies. From the **DDG** and the

CDG a **CFG** with possible concurrent **BBs** is constructed as described in [16].

The data-dependencies analysis of the definition and usage of object members and methods, and array elements are under development. An array image is associated with each created array and every element of the image will store information about the utilisation of the correspondent element of the original array. This information is precious to analyse the possibility of loop parallelization (an important compiler technique since most times the loops included in a specification manipulate arrays).

At the moment, the code segments that degrade the execution performance of the application need to be identified by the user with the help of a profiler tool. A library for inserting tags in the Java source code has been developed, which includes the *Tag* interface with methods to specify code segments (*start*, *end*, *rate*, etc) and two classes that implement it in order to specify timing constraints (*maxtime* and *mintime*). The compiler uses the calls to these methods to identify the user-defined code segments and the constraints specified. The use of this library permits the analysis of the performance of the code between tags in the host machine, by the execution of an interpreter or JIT. The GALADRIEL eliminates all the references to this library in the *bytecodes* of a given method after the segment identification. These segments are translated to a model accepted by the **HLS** system.

4. The Path from *Bytecodes* to Hardware

We opt to translate *bytecodes* to a behavioural VHDL subset, instead of translating them to a tool-dependent intermediate model, to make it acceptable by the majority of the **HLS** systems. The translation is facilitated because at this point the loops were identified and extracted and every *bytecode* instruction stores the actual stack-state. Each primitive Java type (*byte*, *short*, *int*, *boolean*, *long*, *float*, *double*, and *char*) is translated to similar VHDL type or to a VHDL library type declaration.

The **HLS** system to be used must meet the following requirements: start from a standard HDL, perform data-flow analysis, pipelining, allocation, binding, data path synthesis, and controller synthesis.

Consider the following simple segment of code included in a specific application: (b + c) * d + e + f; where all the operands are of type *int* (32 bits). The compiler identifies that all the used variables are defined outside the segment and that the next **BB** depends on the 1st stack operand. Thus, it inserts in the architecture body of the entity the read of 5-32 bit numbers and a 32 bit output write (see Fig. 3 where *Din* and *Dout* are entity ports). The operation scheduling preserves the sequence of used operands in the stack to produce the scheduling with the minimum number of cycles.

At the moment, there are a few restrictions in the allowed translation to VHDL:

- Code segments with manipulation of objects are not considered;
- The exception handling mechanism (try - catch - finally) is not allowed;
- The throw of an exception by any instruction that could do it (like *idiv*) is not implemented by the resulted VHDL description;
- Recursion is not allowed;
- A VHDL entity cannot invoke a SW method.

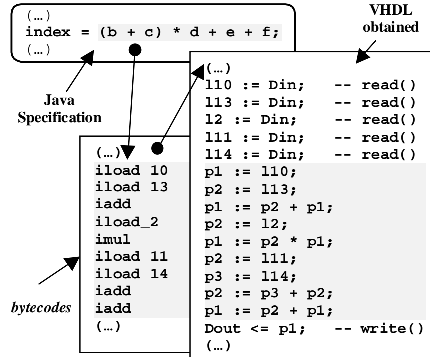

Fig. 3. The translation with read and write operations.

A call to a native method [3] will be inserted in the *bytecodes* every time the SW wants to communicate with the HW. Each platform must have a native communication library, because of the used class representation memory model and architecture discrepancies.

5. An Example

Table 1 presents the results of a filter algorithm (manually translated to Java from the C specification in [17]) executed in a Sun SPARC 10 under SunOS 4.1.4 with the Java interpreter from JDK 1.1.5 [18]. The filter method (*doSobel()*) has 48 lines of Java, compiled to 430 bytes of *bytecodes*. It has 267 instructions grouped in 60 **BBs** (see Fig. 4). This algorithm performs two convolutions (with two nested loops each one) and has more 3 loops. The HW solutions, obtained with the use of the CADDY-II [19] **HLS** system, follows the DMA principle, stores the intermediate arrays in memory, and was mapped to the IMS Sea-Of-gates technology [20]. The back-end logical synthesis phase is done with the Synopsys™ Design Compiler [21].

Solution	Code Segment	Input image (m x n pixels)	
		16 x 16	64 x 64
SW	*doSobel()*	T_{exec} = 3 ms	T_{exec} = 49 ms
	1st Conv.	T_{exec} = 1 ms	T_{exec} = 19 ms
HW @ 25 MHz	1st Conv.	1597 cycles Area = 5,237 eq. Gates T_{exec} = 63.88 μs	51709 cycles Area = 11,911 eq. Gates T_{exec} = 2.07 ms

Table 1. Results of the *doSobel()* method.

The GALADRIEL executes in 11.1s and predicts that the three last loops can be executed in parallel. Moreover if

IEEE COPYRIGHT

we consider the loop inter-iterations parallelization all the iterations of the three loops can be executed in parallel.

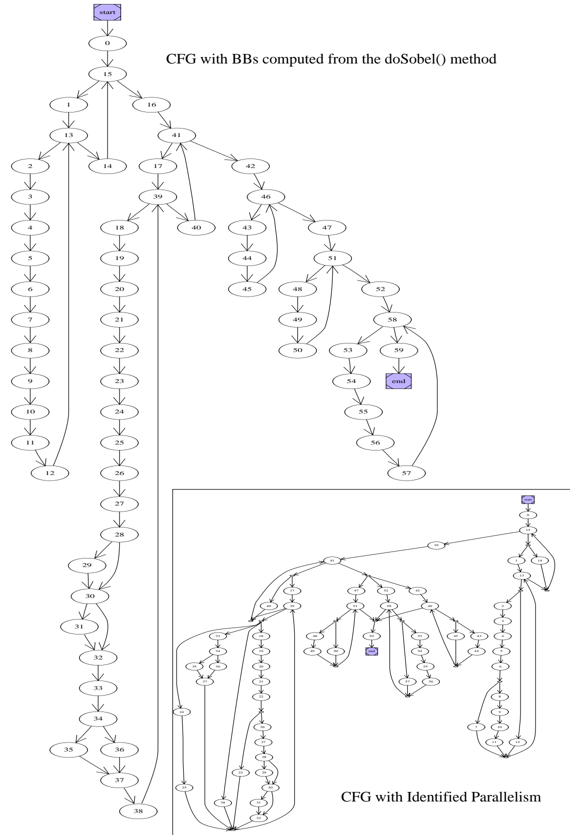

Fig. 4. The basic block's *doSobel()* CFG and the obtained CFG from the identification of the parallelizable BBs (right inferior corner).

6. Conclusions

We have described the techniques used in order to provide an HW performance acceleration of Java *bytecodes*. The developed GALADRIEL compiler allows an efficient front-end to **HLS** systems starting from a system Java specification. It has the capability to determine intra-procedural implicit parallelism, which is an effective way to exploit HW/SW trade-offs, specially when the overall system consists of multi-components, thus having the possibility to map the parallel blocks into independent devices. All the code of the compiler has been developed in Java, assuring therefore its portability, and could be embedded in a Web browser. The current version of the GALADRIEL consists approximately of 7000 lines of Java. This work is one of the first endeavours, known by us, that attempts to use codesign techniques to accelerate Java applications.

The object-oriented specification encourages the use of many methods. Hence an inter-procedural (global) analysis is under development. The migration of methods and/or objects entirely to HW is feasible because typically the fan-in of the methods is small and the object-data can be stored in local memory. Because of the dynamic nature of the language, a static analysis of the memory object allocation like in [2] is performed whenever possible.

The use of the Java *classfile* intermediate representation, makes possible to support multi-language specifications.

References

[1] James Gosling, Bill Joy, and Guy Steele, The Java Language Specification. Addison-Wesley, Reading, Massachusetts, 1996.

[2] R. Helaihel, and K. Olukotun, "Java as a specification Language for HW-SW Systems", In Proc. of the ICCAD'97, San Jose, California, November, 9-13, 1997, pp. 690-697.

[3] Ken Arnold, and James Gosling. The Java Programming Language. Addison-Wesley, Reading, Massachusetts, 1997.

[4] Timothy Cramer, et al., "Compiling Java Just in Time", IEEE Micro, May/June 1997, pp. 36-43.

[5] Zoran Budimlic and Ken Kennedy. "Optimizing Java - theory and practice". Concurrency, Practice and Experience, 9(6): 445-463, 1997.

[6] Michal Cierniak and Wei Li. "Optimizing Java bytecodes". Concurrency, Practice and Experience, 9(6): 427-444, 1997.

[7] Aart J.C. Bik, and Dennis B. Gannon, "javab - a prototype bytecode parallelization tool", Technical Report, Comp. Sci. Dep., Indiana University, USA, 1997, pp. 46.

[8] J. Michael O'Connor and Marc Tremblay, "picoJava-I: The Java Virtual Machine in Hardware", IEEE Micro, Vol. 17, No. 2, March-April 1997, pp. 45-53.

[9] G. De Micheli, Synthesis and Optimization of Digital Circuits, McGraw Hill, 1994.

[10] Rolf Ernst, J. Henkel, T. Benner, "HW-SW Cosynthesis for Microcontrollers", IEEE Design & Test of Computers, Vol. 10, No. 4, December 1993, pp. 64-75.

[11] P. Athanas, and H. Silverman, "Processor Reconfiguration Through Instruction-set Metamorphosis: Architecture and compiler", IEEE Computer, March 1993, vol. 28, no. 3, pp. 11-18.

[12] R. K. Gupta, Co-Synthesis of HW and SW for Digital Embedded Systems, Kluwer Academic Publishers, Boston, MA, 1995.

[13] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. Addison-Wesley, Reading, Massachusetts, 1996.

[14] Alfred V. Aho, Ravi Sethi, and Jefferey D. Ullman, Compilers: Principles, Techniques and Tools, Addison Wesley, 1986.

[15] E. Koutfios, S. C. north, "Editing graphs with DOTTY", User's Manual, version 94b, 1994.

[16] A. V. Chichkov, "HW/SW Co-design Methodology for Custom Computing Machines using Partitioning Based on the Implicit Parallelism", Ph. D. Thesis (in Portuguese), IST (Instituto Superior Técnico), Lisbon, January 1998.

[17] 3rd Annual VIUF Design Contest, 1997. See <http://rassp.scra.org/Fall97_VIUF/Design_Contest/>.

[18] Sun Microsystems™, The Java developer's Kit, 1997. version 1.1. See <http://java.sun.com/products/jdk/1.1/>.

[19] CADDY-II, FZI, Karlsruhe, version 5.10. See <http://www.fzi.de/sim/people/bringmann/caddy/caddy.html>.

[20] IMS, "1.2 µm CMOS Gate Forest Static Cells", 15-Dec-93.

[21] "Design Compiler v3.3a", Synopsys Inc., April 8, 1995.